Corrigé TP11

Ce document présente une proposition de corrigé pour les premières questions du TP11. Bien entendu, le jour d'une réelle épreuve vous n'aurez pas le temps de penser à absolument tout ce que je vais expliquer ici, mais je vous invite à bien étudier ce corrigé de sorte à intégrer quelques astuces de présentation et réflexes d'analyse. Les "points ENS" sont spécifiques à l'épreuve des ENS ; tout le reste est générique.

Point ENS : Sous réserve que l'épreuve d'algorithmique des ENS soit la même en MPI qu'en MP option info, cette dernière consiste en 3h30 de préparation sur machine dans le langage de votre choix parmi un pool de langages autorisés puis de 30min d'exposé devant au moins deux jurys. On vous remettra en début d'épreuve un entier u_0 destiné à être la graine d'un générateur pseudo-aléatoire que vous implémenterez en début d'épreuve de sorte à créer des jeux de tests qui vous sont spécifiques. Les résultats numériques que vous obtiendrez au fil des questions sont à reporter sur une fiche réponse fournie par le concours sur laquelle doit figurer votre nom et votre u_0. Je vous conseille d'écrire ces derniers sur votre fiche réponse avant toute chose.

1. **Point ENS :** Tous les sujets des ENS commencent par cette question. Il faut donc savoir la faire vite et bien et pouvoir expliquer les choix effectués.

   On commence par définir une variable `u0` de manière globale. De cette façon, on pourra écrire toutes nos fonctions en fonction de la variable `u0` et on pourra les tester avec le u_0 exemple donné par l'énoncé et son propre u_0 en changeant juste la valeur de `u0` à un seul endroit.

   ```
   let u0 = 42
   ```

   Décomposer la question : Même si cette question semble simple, il est prudent de la décomposer en deux points afin de ne pas mélanger les deux réductions modulaires successives :

   1) Ecrire une fonction qui permet de calculer u_i étant donné $i \in \mathbb{N}$.
   2) Se servir de cette fonction pour répondre à la question.

   Répondre à la question (1) : Pour le premier point, le piège est de vouloir travailler récursivement. En effet, si on calcule u_i avec une fonction récursive f, on fera a priori i appels à f sur l'entrée i. Or, comme on a bien lu le sujet avant de commencer à implémenter, on a bien vu qu'on aura très fréquemment à calculer des u_i avec parfois des i grands. On voudrait donc que la fonction qui calcule les u_i soit efficace, idéalement que sa complexité soit constante pour tous les i pour lequels on aura besoin de calculer u_i.

   La méthode classique pour ce faire est de calculer une bonne fois pour toutes un tableau contenant u_i en case i pour tous les i jusque $i = N$ avec N suffisamment grand. Le calcul de ce tableau se fait de façon linéaire en N si on le remplit de gauche à droite.

   ```
   let initialiser (u0:int) (n:int) =
     let u = Array.make n 0 in
     u.(0) <- u0;
     for i = 1 to (n-1) do
       u.(i) <- (16807 * u.(i-1)) mod 2147483647
     done;
     u
   ```

   Il ne restera plus qu'à accéder à la bonne case pour calculer rapidement des valeurs de la suite u. Pour déterminer le nombre de u_i à stocker, on observe l'énoncé et on prend un peu de marge.

```
(* tableau initialisé une bonne fois pour toutes *)
let valeurs_memorisees_u = initialiser u0 1500000

let u i = valeurs_memorisees_u.(i)
```

Répondre à la question (2) : Reste à présent à traiter le deuxième point : utiliser la fonction `u` pour répondre à la question posée. On peut se contenter de faire les appels demandés à la fonction `u` mais ce n'est pas très pratique car lorsque vous serez devant le jury, vous ne vous rappelerez probablement plus quelle ligne est sensée répondre à quelle sous question et avec quels arguments. Pour une meilleure présentation, vous pouvez écrire une fonction affichant clairement pour chaque sous question l'entrée demandée par l'énoncé puis le résultat obtenu pour cette entrée :

```
let q1 () = Printf.printf "a = %d, b = %d, c = %d"
((u 10) mod 1234) ((u 10000) mod 1234) ((u 1000000) mod 1234)
```

Ainsi, exécuter `q1 ()` vous permettra d'avoir toutes les réponses à cette question en une fois et de manière lisible.

2. **Comprendre la question :** La première étape face à une telle question est de comprendre ce que veut dire "écrire une machine de Turing". Comme pour un automate fini, une telle machine est entièrement décrite par son nombre d'états et ses transitions : on devine donc qu'il s'agit dans chaque cas de déterminer les transitions de la machine car son nombre d'états est déjà connu par définition. Un coup d'oeil rapide à la réponse type qu'on doit obtenir pour le u_0 exemple de l'énoncé nous confirme cette idée.

   Le piège à présent est d'essayer de calculer les machines $T_{2,2}, T_{2,3}$ et $T_{3,4}$ à la main. C'est une mauvaise idée pour au moins deux raisons :

   - C'est très pénible et on a toutes les chances de se tromper.
   - A terme il faudra automatiser la création de machines $T_{n,t}$; on le sait car on a lu l'énoncé et en particulier la question 3 qui demande de faire des opérations sur 1000 machines de ce type.

   Décomposer la question : On peut décomposer la réponse à cette question en trois étapes :

   1) On va devoir créer des machines : il faut donc construire un nouveau type (ou une struct si vous êtes en C) permettant de représenter de tels objets.
   2) On pourra ensuite écrire une fonction `creer_machine` qui à partir de n, t renvoie $T_{n,t}$.
   3) Enfin, on se servira de cette fonction pour répondre à la question.

   Répondre à la question (1) : Le premier point est absolument critique et le choix de votre type / structure `machine` ne doit absolument pas être fait à la légère car cela aura des répercussions tout au long du sujet. Il est donc impératif d'avoir parcouru l'énoncé pour savoir quelles sont les opérations qu'il faudra fréquemment effectuer avec les objets que vous introduisez.

   Exemple : Si dans le sujet vous fait manipuler des ensembles d'entiers de sorte que l'une des opérations les plus fréquentes soit l'extraction d'un entier au hasard de l'ensemble, il est raisonnable de modéliser un ensemble par une liste. Si le sujet demande en plus de fréquemment tester la taille de l'ensemble, alors vous pouvez opter pour un enregistrement comprenant un champ représentant une liste contenant les éléments et un champ destiné à stocker la taille de la liste : à chaque ajout ou extraction ce champ sera modifié et il permettra d'obtenir la taille de l'ensemble en temps constant et non en temps linéaire. Si le sujet demande plutôt de faire beaucoup d'intersections d'ensembles dont les éléments sont toujours

bornés par M, on pourrait choisir de représenter un ensemble par un tableau de taille M contenant true en case i si et seulement si l'élément i est dans l'ensemble (dans ce cas le calcul d'une intersection se fait linéairement en M) en soulignant nénamoins que pour de petits ensembles et un M grand, cette représentation est coûteuse.

Un autre exemple fréquent intervient lorsque l'énoncé demande de modéliser un graphe : il y a alors un choix à faire entre matrice d'adjacence et listes d'adjacence et généralement une des deux représentations est bien plus pratique à utiliser que l'autre.

Dans notre cas, on s'inspire de la façon dont on a représenté des automates finis. On choisit de stocker les transitions dans une matrice (on pourrait aussi utiliser une table de hachage pour espérer gagner en occupation mémoire mais dans notre cas ça n'a pas grand intérêt car une machine de Turing étant un "automate complet", la matrice représentant la table des transitions sera remplie). En lisant la suite du sujet on se rend compte qu'on nous demandera de suivre des suites de transitions dans une machine de Turing, aussi, il est important que l'opération permettant de déterminer l'action à faire connaissant l'état q dans lequel on est et la lettre x lue sur le ruban soit efficace : avec notre représentation c'est le cas puisqu'elle se fait en temps constant.

```
(* définition d'une MT *)
(* action = état * symbole * direction *)
type action = int * int * int
(* vu leur numérotation il y a juste besoin de connaître le nombre d'états *)
type machine = {nb_etats : int ; delta : action array array}
```

Remarque Ocaml : Il est généralement préférable lorsque vous définissez un type non récursif d'en faire un enregistrement plutôt qu'un tuple : cela clarifie votre code et permet d'accéder facilement à chacun des champs. Cette façon de faire est d'ailleurs la seule possible en C.

Il reste un point à éclaircir pour que notre type machine soit correctement décrit : comment représenter l'état terminal ? Il s'agit de se fixer une convention sur l'entier représentant $\perp$. On peut choisir par exemple -1 ou n où n est le nombre d'états de la machine puisque les "vrais" états d'une machine sont numérotés de 0 à $n-1$. En l'occurrence ce choix n'est pas anondin comme on le verra à la question 5. Dans ce corrigé, on choisit la deuxième option : $\perp$ est représenté par l'entier n.

Répondre à la question (2) : Une fois les choix de structure faits, la fonction creer_machine est immédiate, la seule difficulté étant d'être méticuleux et de ne pas s'embrouiller dans les notations.

```
let creer_machine (n:int) (t:int) =
  let transitions = Array.make n [||] in
  let q' indice =
    if (u (indice+1)) mod n = 0 then n else (u indice) mod n
  in
  let action etat lettre =
       let z = t + 8*etat + 4*lettre in
       ((q' z), (u (z+2)) mod 2, 2*(((u (z+3)) mod 2)) -1)
  in
  for q = 0 to (n-1) do
      transitions.(q) <- [|action q 0; action q 1|]
  done;
  {nb_etats = n ; delta = transitions}
```

Remarque Ocaml : Attention à la façon de créer une matrice en Ocaml ! Array.make n (Array.make n 0) créé un tableau de tableaux dont les lignes sont interdépendantes : si vous modifiez la case i de l'une, vous modifiez la case i de toutes les autres. La raison en est que le Array.make extérieur va se contenter de stocker en case i un pointeur vers le seul et unique tableau créé par le Array.make interne. Ce fonctionnement est d'ailleurs le même avec les listes-Python.

Pour éviter ce désagrément le plus simple est d'utiliser Array.make_matrix. Vous pouvez également faire comme ci-dessus en écrasant chacune des lignes du tableau externe avec un nouveau tableau.

Répondre à la question (3) : Laissé en exercice, inspirez vous des fonctions d'affichages des autres questions.

3. **Décomposer la question :** Comme pour les questions précédentes, il est utile de décomposer la questions en étapes, chacune donnant naissance à une fonction : cela permettra de garder un code lisible et de faciliter son explication :

   1) On va d'abord créer une fonction nb_transitions_terminales destinée à calculer le nombre de transtions menant vers $\perp$ dans une machine donnée.

   2) La fonction compte_machines_q3 permettra, à partir d'un entier k, de déterminer quelles sont les machines dans l'ensemble proposé qui ont un nombre de transitions terminales égal à k.

   3) Il ne restera plus qu'à gérer l'affichage de la réponse à la question.

Répondre à la question (1) : Une transition est terminale si et seulement si elle mène vers l'état $\perp$. Il suffit donc de parcourir toutes les transitions et d'identifier celles pour lesquelles l'état d'arrivée porte le numéro qu'on a conventionnellement attribué à $\perp$.

```
let nb_transitions_terminales (m:machine) =
  let n = m.nb_etats in
  let nb_terminales_etat (q:action array) :int =
    let q0, _, _ = q.(0) and q1, _, _ = q.(1) in
    match q0, q1 with
    |u,v when (u = n && v = n) -> 2
    |u, _ when (u = n) -> 1
    |_, v when (v = n) -> 1
    |_, _ -> 0
  in let nb = Array.map nb_terminales_etat m.delta
     in Array.fold_left (+) 0 nb
```

Remarque Ocaml : Pour récupérer les champs d'un tuple, la façon la plus simple est de le dépaqueter comme en Python (exemple ci-dessus). On peut aussi se servir d'un filtrage mais c'est souvent plus lourd.

Remarque Ocaml : Dans cette fonction, j'ai adopté un style fonctionnel (nb est un tableau contenant en case q le nombre de transitions terminales depuis l'état q et il ne reste donc plus qu'à sommer les cases de ce tableau pour avoir la réponse). Vous pouvez également vous servir d'un compteur et d'une double boucle pour pour compter les transitions terminales si vous le souhaitez.

Répondre à la question (2) : Pas de difficulté : il suffit d'utiliser la fonction précédente afin d'isoler les machines de l'ensemble $\{T_{4,t} \,|\, 0 \leq t < 1000\}$ qui ont le bon nombre de transitions terminales. Dans un style impératif on aurait par exemple :

```
let compte_machines_q3 (k:int) =
  let nb = ref 0 in
  for t = 0 to 999 do
    let m = creer_machine 4 t in
    if nb_transitions_terminales m = k then
      nb := !nb +1
  done;
  !nb
```

Dans un style plus fonctionnel cela pourrait donner :

```
let compte_machines_q3_rec (k:int) =
  let rec liste_machines (n:int) =
    (* calcule la liste des machines T_{4,t} pour t <= n *)
    if n = 0 then [creer_machine 4 0]
    else let queue = liste_machines (n-1) in (creer_machine 4 n)::queue
  in let l = liste_machines 999
     in List.length
          (List.filter (fun m -> (nb_transitions_terminales m = k)) l)
```

Comme on a lu le sujet, on a remarqué que ce dernier demande de compter les machines ayant certaines propriétés dans plusieurs questions différentes d'où le choix d'inclure le numéro de la question dans le nom de la fonction ci-dessus et pas juste de l'appeler compte_machine.

Répondre à la question (3) : Un affichage clair peut être celui-ci :

```
let q3 () = Printf.printf "k = 0 : %d transitions terminales,
                           k = 1 : %d transitions terminales,
                           k = 3 : %d transitions terminales"
              (compte_machines_q3_rec 0)
              (compte_machines_q3_rec 1)
              (compte_machines_q3_rec 3);;
q3 ();;
```

4. **Décomposer la question :** La question demande d'étudier un ensemble de machines et d'isoler celles pour lesquelles l'exécution à partir de la configuration initiale finit en moins d'un certain nombre d'étapes. Pour ce faire, on va vraisemblablement avoir besoin de simuler l'exécution d'une machine de la même façon qu'on aurait simulé la lecture d'un mot dans un automate fini. Cette simulation consiste à passer de configuration en configuration en respectant les transitions. La question orale 3 nous indique ainsi la première étape permettant de décomposer cette question en blocs simples :

   1) On commence par déterminer une structure adaptée pour représenter une configuration.
   2) On écrit ensuite une fonction etape prenant en entrée une machine et une configuration et renvoyant la configuration suivant celle en entrée. C'est la brique élémentaire permettant de calculer des suites de configurations.
   3) On écrit ensuite une fonction arret_k déterminant si une machine s'arrête en moins de k étapes.
   4) On pourra alors compter le nombre de machines parmi $\{T_{4,t} \,|\, 0 \leq t < 1000\}$ s'arrêtant en moins de k étapes à l'aide de nb_machines_arret_k.
   5) Il ne restera plus qu'à régler les questions d'affichage de la réponse.

Répondre à la question (1) : L'énoncé précise qu'on n'utilisera qu'une partie finie du ruban. Il est ainsi justifié de modéliser le ruban par un tableau (dont on sait que la taille sera majorée par 501). Pour caractériser une configuration, on a besoin, en plus de l'état du ruban, de connaître la position de la tête de lecture sur ce ruban et l'état du système. On propose donc le type suivant :

```
type configuration = {bande : int array;
                      mutable tete : int;
                      mutable etat : int}
```

Comme on a lu l'énoncé, on a constaté que connaître exactement la suite des configurations lors d'un calcul ne nous intéresse jamais ; ce qui est recherché c'est quelles sont les tailles de ces suites de configurations. Afin d'éviter de stocker des listes ou des tableaux de configurations lors d'une exécution (ce qui serait coûteux en mémoire et ne nous intéresse pas car seule compte la taille de ces structures vu l'énoncé), on décide donc lors d'un calcul à partir d'une configuration c de modifier c au fur et à mesure de l'exécution : c'est pourquoi on rend les champs tete et etat mutables.

On en profite pour écrire une petite fonction renvoyant la configuration initiale de toutes les machines qu'on va étudier. Attention, la position de la tête doit être au milieu du tableau représentant le ruban.

```
(* Il faut un tableau symétrique autour d'une valeur de taille au moins 501 *)
let config_initiale () = {bande = Array.make 501 0;
                          tete = 250;
                          etat = 0}
```

Répondre à la question (2) : La fonction etape consiste juste à modifier la configuration en entrée selon la transition effectuée. On anticipe l'étape suivante en introduisant une exception destinée à signaler que la configuration obtenue est terminale (de sorte à ne pas poursuivre le calcul une fois qu'on a atteint l'état terminal).

```
exception Fin of configuration

let etape (m:machine) (c:configuration) =
  let n = m.nb_etats in
  let lettre_lue = c.bande.(c.tete) in
  let etat_courant = c.etat in
  let (etat_suivant, lettre_ecrite, direction) =
     m.delta.(etat_courant).(lettre_lue) in
  if etat_suivant = n then raise (Fin c);
  c.etat <- etat_suivant;
  c.bande.(c.tete) <- lettre_ecrite;
  c.tete <- c.tete + direction
```

Répondre à la question (3) : Une fois la fonction etape écrite, arret_k n'est pas très compliquée à implémenter. Il faut juste être prudent : dans une exécution de longueur k, il y a k configurations donc $k-1$ transitions. On tente donc de faire au plus $k-1$ appels à etape et si une itération lève l'exception Fin c'est que l'exécution a terminé avant d'atteindre la borne fixée sur la taille de l'exécution.

Ceci n'est qu'une parmi de nombreuses façons de gérer une interruption prématurée de l'exécution. On peut plus simplement se servir d'une conditionnelle pour distinguer le cas particulier où faire une étape amène dans l'état terminal et gérer un compteur incrémenté à chaque étape qu'on interrompera si il dépasse $k-1$.

```
let arret_k (m:machine) (k:int) =
  let config = config_initiale () in
  try
    for i = 1 to (k-1) do
      etape m config;
    done;
    false
  with
  |Fin _ -> true
```

Répondre à la question (4) (5) : Les idées sont très similaires à celles de la question 3 :

```
let nb_machines_arret_k (k:int) =
  let compteur = ref 0 in
  for t = 0 to 999 do
    if arret_k (creer_machine 4 t) k then
      compteur := !compteur +1
  done;
  !compteur
```

On aurait pu pour nb_machines_arret_k utiliser un style fonctionnel, à la façon de compte_machines_q3_rec. Faites le en exercice. Pour l'affichage, la façon de faire ci-dessous évite d'avoir à recopier ce qu'il faut afficher pour chacun des trois cas, c'est un peu plus élégant :

```
let q4 () =
  let affiche (k:int) =
    Printf.printf "k = %d / nombre de machines = %d\n"
      k (nb_machines_arret_k k)
  in List.iter affiche [2;10;100]
;;
q4();;
```

5. **Comprendre la question :** La difficulté de cette question est que l'énoncé donne (volontairement ?) une description très alambiquée de $L(T)$. Que veut dire pour un état q d'être dans $L(T)$? Tout simplement que quelques soient les transitions suivies à partir de q, on n'atteindra jamais l'état $\perp$. Plus concrètement :

$$L(T) = \{\text{états de T qui ne permettent pas d'atteindre } \perp\}$$

A ce stade, on peut donc reformuler la question en termes d'accessibilité dans un graphe : déterminer $L(T)$ revient à trouver les états q tels que $\perp$ n'est pas accessible depuis q dans le graphe G sous-jacent à la machine T. Pour trouver ces états q, il suffit de transposer G et de faire un parcours depuis le sommet correspondant à $\perp$: les états de $L(T)$ sont exactement ceux qu'on ne peut pas atteindre via ce parcours.

Et nous voilà ramenés à écrire un parcours de graphe ! Cela permet du même coup de répondre facilement à la question orale 5 : la complexité du calcul de $L(T)$ est celle d'un parcours dans un graphe ayant $2n$ arcs et n sommets dont est en $O(n)$ avec n le nombre d'états dans la machine.

Décomposer la question : Après avoir compris que l'enjeu est d'écrire un parcours de graphe, on peut décomposer le travail comme suit :

1) Ecrire une fonction transposer prenant en entrée une machine et transposant le graphe sous-jacent.

2) Ecrire une fonction calcule_l prenant en entrée une machine T à n états et renvoyant un tableau a $n+1$ cases contenant true en case i si et seulement si i est dans $L(T)$. C'est là qu'on est content d'avoir choisi n pour représenter l'état terminal et pas -1 !

3) Ecrire deux petites fonctions de post-traitement contient_0 et cardinal_moitie destinées à savoir si 0 est dans $L(T)$ et si le nombre d'états dans $L(T)$ vaut $n/2$ avec n le nombre d'états de T.

4) Gérer l'affichage des réponses à la question.

Répondre à la question (1) : Pas de difficulté : on parcourt les arcs via la table des transitions de la machine et on les inverse. Ce faisant, on est en fait en train de construire un multi-graphe et pas un graphe. Par exemple, si lire un 0 dans l'état 3 fait passer dans l'état 4 et pareil si on lit un 1 alors dans la liste d'adjacence de 4 dans le "graphe" transposé, il y aura deux occurrence de 3. Est-ce grave ? Vu que ça n'impacte pas la correction d'un parcours, pas du tout.

```
let transposer (m:machine) =
  let n = m.nb_etats in
  let gt = Array.make (n+1) [] in
  for q = 0 to (n-1) do
    for x = 0 to 1 do
      let q', _, _ = m.delta.(q).(x) in
      gt.(q') <- q::gt.(q')
    done;
  done;
  gt
```

Répondre à la question (2) : La fonction calcule_l peut être implémentée en utilisant un parcours en largeur ou en profondeur au choix. J'opte pour le second. A la fin de l'appel de visite n, vus indique quels sont les états accessibles depuis $\perp$: on veut au contraire les états non accessibles depuis $\perp$ d'où la dernière ligne du code.

```
let calcule_l (m:machine) =
  let n = m.nb_etats in
  let gt = transposer m in
  let vus = Array.make (n+1) false in
  let rec visite (q:int) =
    if not vus.(q) then
      begin
        vus.(q) <- true;
        List.iter visite gt.(q)
      end in
  visite n;
  Array.map not vus
```

Répondre à la question (3) (4) : Pour contient_0, c'est immédiat.

```
(* indique si la case 0 d'un tableau contient vrai *)
let contient_0 (t:bool array) = t.(0)
```

Pour cardinal_moitie il faut faire attention : cette fonction est destinée à être utilisée sur un tableau de taille $n+1$ avec n le nombre d'états d'une machine de Turing. On veut donc vérifier si le nombre de cases du tableau (de taille n') contenant true vaut $(n'-1)/2$.

```
(* indique si un tableau de taille n+1 contient n/2 éléments valant vrai *)
let cardinal_moitie (t:bool array) =
  let n = (Array.length t) -1 in
  let nb_vrais = ref 0 in
  for i = 0 to (n - 1) do
    if t.(i) then incr nb_vrais
  done;
  2* !nb_vrais = n
```

Pour l'affichage, on adopte le style de la question 4. On aurait même pu décomposer encore cette fonction en une fonction qui calcule (a, b) étant donnés (n, M) puis une fonction qui ne fait que l'affichage stricto-sensu.

```
let q5 () =
  let affiche ((n,m):int*int) =
    let a = ref 0 and b = ref 0 in
    for t = 0 to (m-1) do
      let t = calcule_l (creer_machine n t) in
      if contient_0 t then incr a;
      if cardinal_moitie t then incr b
    done;
    Printf.printf "(%d,%d) : a = %d, b = %d\n" n m !a !b
  in List.iter affiche [(4,1000);(8,10000);(32,100000)]
;;

q5();;
```

Point ENS : A partir d'un certain point dans un sujet ENS (parfois dès le début) les tests qui vous sont demandés par les questions de programmation sont calibrés de telle sorte à ce qu'au moins un pousse votre machine à ses limites. Généralement, un code même un peu maladroit arrivera à produire une réponse pour les deux premières valeurs test mais un code non optimisé ne permettra pas d'obtenir une réponse en temps raisonnable pour la dernière.